Sharing Tutorial Chennai

Tutorial on Sharing and Interacting in Sage

This Sage worksheet is for the course in Sage and programming at the Institute of Mathematical Sciences in Chennai, circling around Sage Days 60. It is partly based on materials used in the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071).

Writing code that powerfully constructs examples or performs mundane calculations is fun and empowering. But perhaps it's even better if we can share and share alike? Maybe that includes allowing others to interact with our code. This tutorial will talk about a variety of ways to share resources and ideas, including via interactive capabilities of Sage.

Sharing your Code

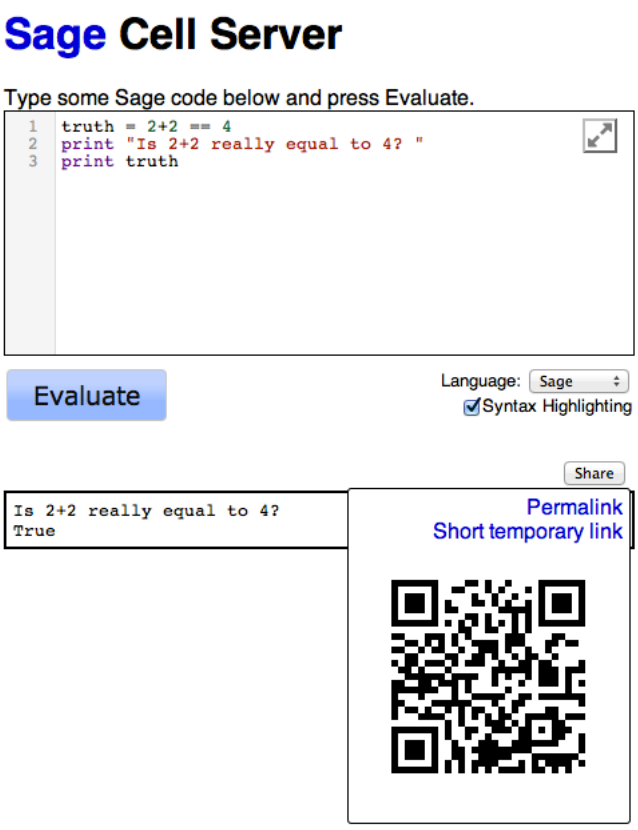
There are several ways to share code with others in Sage interfaces. We'll talk about them one by one, just as we talked about the ways to use Sage on the first day.

- The Sage cell server
- Sage Math Cloud
- The command line Sage
- The Sage notebook (where this tutorial lives)

Sage cell server

You may have noticed that I've provided links on occasion to Sage cell instances. How did I do that?

The key is the "Share" button just to the upper right of your computation.

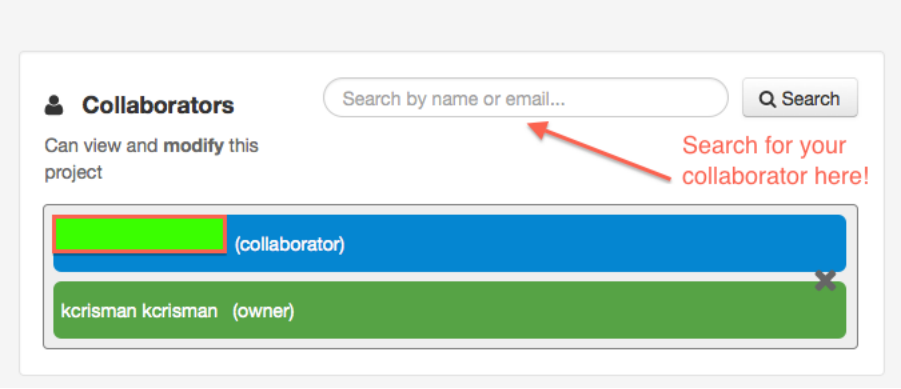


You can cut and paste the link that appears in your browser window and put it in an email, link it to some text or a picture on a website, or whatever. The QR code works too!

The only difference between the permalink types is that the first kind might be very long, as it is an encoded (not encrypted) version of the code itself, but the (shorter) second kind could be purged from a database if the server is getting full or something. In practice, they are both useful.

Sage Math Cloud

If your collaborator is on SMC, you can easily collaborate in real time on code and other files. Just go to the settings for any given project, and search for your collaborator there.



Now you've shared it!

However, it is not currently possible to share with the outside world on Sage Math Cloud.

Sage Command Line

You might think it would be hard to share something from your own computer, and you would be right. Except... you can share *files* you create on your computer, via email or the web.

So imagine you were solving one of the problems from the last homework set, and came up with this code to start with, wanting feedback.

```
def get_consecutive_primes(a,b):
    formula(x) = x^2 + a*x + b
    if not is_prime(formula(0)):
        print formula(x)
        print -1
    else:
        n = 0
        while is_prime(formula(n)):
            n += 1
        print formula(x)
        print n
```

```
get_consecutive_primes(1,41)
```

```
    x^2 + x + 41
    40
```

```
get_consecutive_primes(1,42)
```

```
    x^2 + x + 4
    -1
```

This uses yet *another* control mechanism, called a "while loop", as well as some additional slightly new syntax. You may want to read up on them.

However, the point is that I want for my friend to check out whether it works or could be improved. Or maybe I just need help trying a and b that might work.

In this case, I can cut and paste my code to a file and send that out into the world, post it on a webpage, etc. Be sure to save the file as "my_code.sage". It is not Python, not just text, but a Sage script. This is important because Sage will know that ".sage" implies it needs to load all the Sage-specific things like the integer ring, predefining the variable "x", and so forth.

Then anyone else you give it to can run this file as they would before. They can cut and paste it, or...

Loading and attaching

Rather than running the file, one can even load the file directly in. The function above is not predefined in Sage, but loading this file (with this syntax to the *exact* location in my computer) allows me to use it right away. Here, the file I saved it in is called "gcp.sage".

```
dhcp79:Downloads karl.crisman$ sage
```

```
Sage Version 5.12, Release Date: 2013-10-07
Type "notebook()" for the browser-based notebook interface.
Type "help()" for help.
```

```
sage: load("/Users/karl.crisman/Downloads/gcp.sage")
sage: get_consecutive_primes(1,41)
x^2 + x + 41
40
sage:
```

The command "attach" is very similar, except it monitors the file for changes and updates accordingly.

Warning: Don't save your scripts as Python ".py" files unless you know what you are doing!

This proves to be very useful for contributing code to Sage, and we will look at this next time; when you are starting it wil not always be clear what needs to be changed for Python. So just use ".sage" for now.

Finally, you can also save and load complete sessions and share these, but that is beyond the scope of these tutorials.

Sage notebook

Sharing notebook files is easy to do, in three ways.

1. First, you can always download your worksheet. This can be done from the worksheet itself, or from the master list of worksheets. This will save to a ".sws" file (or a zip of several such files) and you can email that, post it on a webpage, etc. This is not incredibly efficient, but as long as there isn't a lot of back and forth is quite acceptable. It is a good way to share a complex worksheet of research code.
2. If your collaborators are registered on the same Sage notebook server as you, you can also share with them there.
3. If your server allows it, you can *publish* your worksheets and allow them to be viewed with the rest of the world (though not executed, at least not without downloading a copy).

Let's demonstrate each of these things now.

My own experience is that sharing is good for asynchronous work together. Especially if you are doing a large computation, and then want to make sure someone else has direct access to it, this is helpful.

Publishing is very useful for students who are on the same server as you, or for sharing resources with the world - except you have to watch for spam, since these are arbitrary webpages and might have untrustworthy content if you let just anyone create and then publish a worksheet from your server.

Interestingly, you can also do the same loading of .sage files in the notebook. Below, we'll just make sure that our function for consecutive primes no longer exists.

```
reset('get_consecutive_primes')
```

```
get_consecutive_primes(1,41)
```

```
Traceback (click to the left of this block for traceback)
...
NameError: name 'get_consecutive_primes' is not defined
```

Now I'll load the same file, which I already uploaded to the notebook using the "Data" pulldown menu at the top of the worksheet.

```
load(DATA+'gcp.sage')
```

```
get_consecutive_primes(1,41)
```

```
x^2 + x + 41
40
```

In summary, there are MANY ways to share your Sage code!

Interacting with your code

Of course, sharing can get tedious too, especially if it's hard to interact with your code. Luckily, Sage has a really nice way to *interact* with your computations that is worth spending a little time exploring.

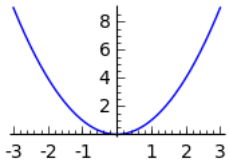
We'll slowly build up this interactivity via examples; for many, many more examples, visit the Sage wiki http://wiki.sagemath.org/interact (type "sage interact" into Google), the code snippet sharing site for Sage interact.sagemath.org, and the interact documentation.

First example

For our first example, let's go back from the world of higher math to just something very simple - plotting graphs. The reason for this is one can see one's work more obviously in this case.

Let's start by getting the commands for some output for a simple plot. We use the "figsize" keyword because we zoom in so much for the projector.

```
plot(x^2, (x,-3,3), figsize=2)
```

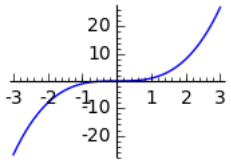


Here comes the part that is universal: We abstract out the parts we want a user to be able to interactively change.

First, let's just let the user change the function, and call that a variable f.

```
f = x^3
```

```
plot(f, (x,-3,3), figsize=2)
```



This was important because it allowed you to step back and think about what you would really be doing.

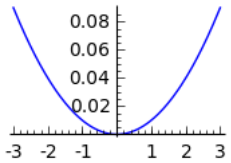
Now for the technical part. We first make this a "def" function. Note also that we give the variable a default value of x^2.

- (The "show" or "print" is needed since the output is not automatically printed from within a function.)

```
def myplot( f = x^2 ):
    show( plot(f,(x,-3,3),figsize=2) )
```

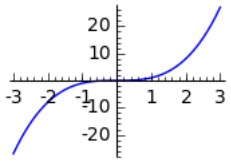
Let's test the def function myplot by just calling it.

```
myplot(0.01*x^2)
```



If we call it with a different value for f, we should get a different plot.

```
myplot(x^3)
```



So far, we've only defined a new function, so this was technically review. To make a control to enter the function, we just preface the function with @interact.

```
@interact
def myplot(f=x^2):
    show(plot(f,(x,-3,3),figsize=2))
```

What if I now want the user to interact with more stuff? We can go ahead and replace other parts of the expression with variables, and I'll do so for the endpoints of the plot.

- Minor point - I'm also changing the function name to the underscore symbol "_" ; this is just a convention for throw-away names that we don't care about.

```
@interact
def _( f=x^2, a=-3, b=3 ):
    show(plot(f,(x,a,b),figsize=2))
```

If we pass ('label', default_value) in for a control, then the control gets the label when printed. Here, we've put in some text for all three of them. Remember that the text must be in quotes! Otherwise Sage will think that you are referring (for example) to some variable called `lower`, which it will think you forgot to define.

```
@interact
def _(f=('$f$',x^2), a=('lower', -3), b=('upper',3)):
    show(plot(f,(x,a,b),figsize=2))
```

So this is kind of nice. But what else can I do with it?

Going deeper

Well... I can search for theorems!

```
@interact
def power_table_plot(p=(7,prime_range(50))):
    P=matrix_plot(matrix(p-1,[mod(a,p)^b for a in range(1,p) for b in srange(p)]),cmap='jet',figsize=4)
    show(P)
```

This is actually complete information about the behavior of powers in finite fields of prime order in *color-coded* format. The $a-1$ row and b column gives the color corresponding to $(a-1)^b$ mod (p). That means the first (0th) column is the color for 1 and the second (1th) column gives the colors of each element of $\mathbb{Z}_p$. For instance, $(3,4)$ corresponds to 2^4 mod (7) in the initial example.

So... what theorems can you find?

Control types

In this second example, instead of having a box, I had a drop-down list. If you look carefully, that is because I used "prime_range(50)" as the default value, which gives a list of primes less than 50; lists turn into drop-down lists. Sage also has:

- input boxes
- sliders
- range sliders
- checkboxes
- selectors (dropdown lists or buttons)
- grid of boxes
- color selectors
- plain text

The documentation for "interact" is very, very thorough, though you have to really try out the options for inputs to see how to use them.

So going back to our function about primes from a polynomial, at least it would be more fun to look with this interactive output.

```
@interact
def _(a=range(1,100),b=range(1,100)):
    return get_consecutive_primes(a,b)
```

Let's try to customize our initial plotting examples just a little bit more. Recall that we had boxes for inputting the function and the endpoints we wanted to plot.

```
@interact
def _(f=('$f$',x^2), a=('lower', -3), b=('upper',3)):
    show(plot(f,(x,a,b),figsize=2))
```

Maybe I want to allow different styles for the line. Looking at the documentation for plotting, I see I have a few options. Let's give the user the ability to use them.

```
@interact
def _(f=('$f$',x^2), a=('lower', -3), b=('upper',3),s=('line style',['-','--',':','']),):
    show(plot(f,(x,a,b),figsize=2,linestyle=s))
```

Or we could add a way to customize how much we want to see of the plot along the vertical axis.

```
@interact
def _(f=('$f$',x^2), a=('lower', -3), b=('upper',3),s=('line style',['-','--',':','']),vert=range_slider(-10,10,1,
(0,10))):
    show(plot(f,(x,a,b),figsize=2,linestyle=s,ymax=vert[1],ymin=vert[0]))
```

In the Sage Math Cloud and the Sage cell server you can even nest interacts, but the syntaxes are slightly different and are not available in the notebook, so I won't cover that here.

A Gallery

We could spend a whole hour just learning about all the options for these "interacts". But I don't think that would be a good use of our time; instead, I want to give you a gallery of what you can do, in the hope that it will inspire you in the search for theorems.

The next one is a visualization of a key part of a proof of quadratic reciprocity.

```
var('x,y')
@interact
def _(p=(11,prime_range(3,100)),q=(7,prime_range(3,100))):
    E = [2,4..p-1]
    plot4 = plot((q/p)*x,(x,0,p),linestyle='--')
    plot3 = line([[0,0],[p,0],[p,q],[0,q],[0,0]],rgbcolor=(1,0,0))
    plot2 = line([[0,0],[(p-1)/2,0],[(p-1)/2,(q-1)/2],[0,(q-1)/2],[0,0]],color='green')
    grid_pts_1 = [[i,j] for i in [1..p] for j in [1..q]]
    grid_pts_2 = [[i,j] for i in [1..(p-1)/2] for j in [1..(q-1)/2]]
    plot_grid_pts = points(grid_pts_1,rgbcolor=(0,0,0),pointsize=10)
    lattice_pts1 = [coords for coords in grid_pts_1 if (coords[0]*q-coords[1]*p>0 and coords[0]<p and coords[0] in
E)]
    lattice_pts2 = [coords for coords in grid_pts_2 if (coords[0]*q-coords[1]*p>0 and coords[0]>p/2)]
    num1, num2 = len(lattice_pts1), len(lattice_pts2)
```

```
    if len(lattice_pts1)!=0:
        plot_lattice_pts1 = points(lattice_pts1, rgbcolor = (0,0,1),pointsize=20)
    else:
        plot_lattice_pts1 = Graphics()
    if len(lattice_pts2)!=0:
        plot_lattice_pts2 = points(lattice_pts2, rgbcolor = (0,.5,0),pointsize=20)
    else:
        plot_lattice_pts2 = Graphics()
    show(plot2+plot3+plot4+plot_grid_pts+plot_lattice_pts1,xmax=p,ymax=q,ymin=0,figsize=4)
    forms = '$'+'+'.join(['\left\lfloor\\frac{%s\cdot %s}{%s}\\right\\rfloor'%(q,e,p) for e in E])+'$'
    html("The blue dots represent "+forms)
    forms2 = '$'+'+'.join(['\left\lfloor\\frac{%s}{%s}\\right\\rfloor'%(q*e,p) for e in E])
    forms3 = '+'.join(['%s'%(floor(q*e/p)) for e in E])+'=%s\equiv%s\\text{ mod }(2)$'%(sum([floor(q*e/p) for e in
E]),sum([floor(q*e/p) for e in E])%2)
    html("This simplifies to "+forms2+'='+forms3)
```

Or, we could try to see how close different approximations to the prime counting function $\pi(x)$ are.

```
@interact
def _(m=3,n=100,auto_update=False):

show(plot(prime_pi,m,n,color='black',legend_label='$\pi(x)$')+plot(x/ln(x),m,n,color='red',legend_label='$x/\ln(x)$')+
```

Here I used the "auto_update=False" keyword to get a convenient way to delay updating. This becomes really important with matrices!

In fact, matrices as input to your function are automatically converted to a grid of input boxes, and in this case we almost certainly want to prevent auto-updating.

```
@interact
def _(m=('matrix', identity_matrix(3)),auto_update=False):
    EV = m.eigenvalues()
    show(point([CC(ev) for ev in EV],size=30),figsize=4)
```

We can have fun too.

```
var('t')
@interact
def _(A=matrix(RDF,[[1,0],[0,1]]),auto_update=False):
    pll=A*vector((-0.5,0.5))
    plr=A*vector((-0.3,0.5))
    prl=A*vector((0.3,0.5))
    prr=A*vector((0.5,0.5))
    left_eye=line([pll,plr])+point(pll,size=5)+point(plr,size=5)
    right_eye=line([prl,prr],color='green')+point(prl,size=5,color='green')+point(prr,size=5,color='green')
    mouth=parametric_plot(A*vector([t, -0.15*sin(2*pi*t)-0.5]), (t, -0.5,
0),color='red')+parametric_plot(A*vector([t, -0.15*sin(2*pi*t)-0.5]), (t,0,0.5),color='orange')
    face=parametric_plot(A*vector([cos(t),sin(t)]),
(t,0,pi/2),color='black')+parametric_plot(A*vector([cos(t),sin(t)]),
(t,pi/2,pi),color='lavender')+parametric_plot(A*vector([cos(t),sin(t)]),
(t,pi,3*pi/2),color='cyan')+parametric_plot(A*vector([cos(t),sin(t)]),(t,3*pi/2,2*pi),color='sienna')
    P=right_eye+left_eye+face+mouth
    html('smiley guy transformed by $A$')
    P.show(aspect_ratio=1,figsize=4)
```

Here is something that looks like fun, but is also related to dynamical systems.

```
def Newton_Graph(a_function, guess, iterations = 5):
    '''
    Returns a graphics object of a plot of a_function and Newton-Raphson method tangent lines.

    INPUT:
        a_function: a (callable) function of one variable
        start: the starting value of the iteration
        iterations: (optional) the number of iterations to draw
        xmin: (optional) the lower end of the plotted interval
        xmax: (optional) the upper end of the plotted interval

    EXAMPLES:
        sage: f = lambda x: x^3-3*x^2+2*x
        sage: show(Newton_Graph(f,.5))
    '''
    iter_list = [[guess,0]]
    f(x)=a_function(x)
```

```
    deriv(x)=f.derivative()
    approx(x)=x-f(x)/deriv(x)
    input = guess
    for i in range(iterations):
        iter_list.append([input,f(input)])
        input=approx(input).n()
        iter_list.append([input,0])
    approx_tangents = line(iter_list,rgbcolor=(1,0,0))
    xmin=min([point[0] for point in iter_list])-1
    xmax=max([point[0] for point in iter_list])+1
    ymin=min([point[1] for point in iter_list])-.5
    ymax=max([point[1] for point in iter_list])+.5
    basic_plot = plot(a_function, xmin, xmax, color='blue',ymin=ymin,ymax=ymax)
    P=basic_plot + approx_tangents
    return P

def cubic_approx(guess, intercept=0, iterations = 5):
    '''
    Returns the Newton-Raphson zero approximation graph for x^3-3*x^2+2*x+intercept, from the given starting
guess.

    INPUT:
        start: the starting value of the iteration
        mask: (optional) the number of initial iterates to ignore
        iterations: (optional) the number of iterations to draw, following the masked iterations

    EXAMPLES:
        sage: P=cubic_approx(.5,3); show(P)
    '''
    f(x)=x^3-3*x^2+2*x+intercept
    return Newton_Graph(f,guess,iterations=iterations)
```

```
@interact
def _(guess=input_box(RR(.55),label='start guess'),intercept=slider(-2,2,.5,1),iterations=slider(1,20,1,5)):
    P = cubic_approx(guess,intercept=intercept,iterations=iterations)
    root = P[1].xdata[-1]
    show(P,figsize=4)
    html('root approximation $\\approx%f$'%root)
    html('$f(x)=x^3-3x^2+2x+%f'%intercept)
```

Lest you think this is just for visualizing easy stuff, you can interact for *anything* in Sage. Here is the parametrized set of graphs of the image under the Riemann zeta function in the complex plane of vertical lines in the complex plane. Somehow the only time it goes through the origin is when the line is the $1/2$ line.

```
@interact
def _(sig=slider(.001, .999, .001, 0.5, label='$\sigma$')):
    end=30
    p = parametric_plot((lambda t: zeta(sig+t*i).real(),lambda t: zeta(sig+t*i).imag()),
(0,end),rgbcolor=hue(0.7),plot_points=300)
    q = complex_plot(zeta,(0,1),(0,end),aspect_ratio=1/end)+line([(sig,0),(sig,end)],linestyle='--')
    show(graphics_array([p,q]),figsize=[5,3])
```

Sage has many interacts built in as well, especially for pedagogical purposes. I like this one and figure there is quite a bit of mathematics left to discover in it.

```
interacts.algebra.polar_prime_spiral()
```

range (1, 1001)

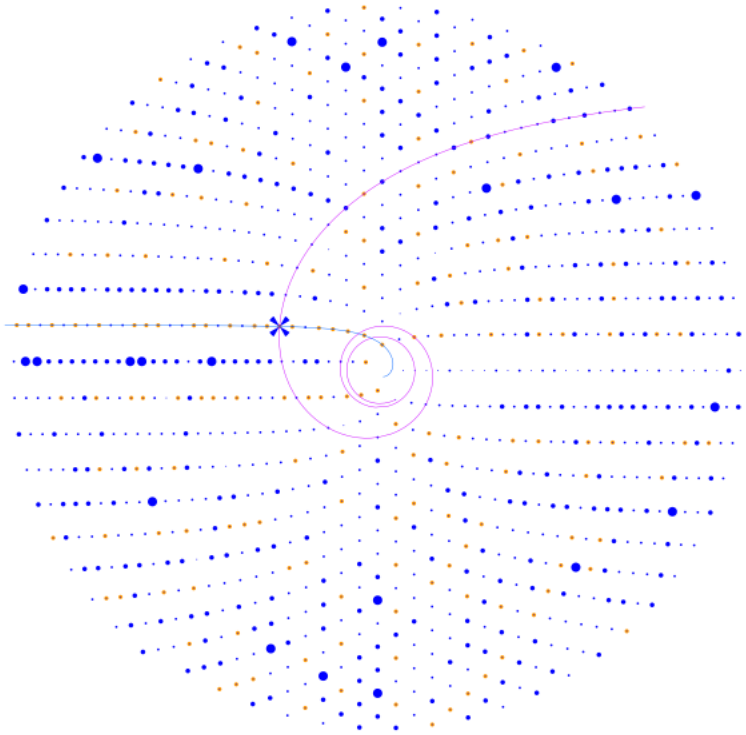
show_factors ☑

highlight_primes ☑

show_curves ☑

number n 89

dpi 100



As mentioned at the beginning, there are many more places to look for examples, starting with the Sage wiki and interact.sagemath.org.

Sharing and Interacting

But the best of both worlds is to share your code *and* let people interact with it! And it turns out that one good way to do that is to enable Sage cell instances *embedded in your web pages*.

What do I mean by that? One example might be to allow graph calculations on your home page.

Another might be to give your students lecture notes that let them calculate homework in their hostel room.

Since this is not a course in web design or HTML, I won't share the full details. But the basic instructions are surprisingly easy to follow without knowing any HTML. Enough so that I can do it, anyway.

More ways to interact

Whether you have questions or like to help others get better at Sage, there are other ways to interact and share your knowledge.

- Sage has a large variety of email lists on Google groups.
  - Sage-support is a place to discuss and have questions asked about any help needed, trivial or not
  - Sage-devel discusses general development direction issues.
  - There are a number of research-oriented lists, such as sage-combinat-devel and sage-nt, as well.
- There are also places to ask questions in a more Q&A format.
  - Ask.sagemath.org is the one-stop-shop for questions, and likely many questions you have are already answered there. But you can answer for others too!
  - Various other Q&A sites have Sage questions, particularly stackoverflow but also sometimes math-oriented sites.
- Finally, you could share your love of Sage via social media!
  - Sage is on Facebook.
  - Sage is on Google Plus.

Homework

1. If you have a question that you could ask online or a Google group it would make sense to join, do so!
2. If you are on FB or G+, you could like/+1 Sage.
3. Look at a page with Sage cells embedded. Read the "View source" of the page. Do not try to make a new one, just try to see what lines seem to be Sage-related. It isn't much, is it?
4. Take some code or a worksheet you are working on and download it. If you have someone you can share it with using one of the mechanisms outlined here, do it!
5. Take some easy computation you did earlier in the course and make it interactive. This is the primary homework question. Here are some steps.
    1. First define a function with the computation in it.
    2. Then make just that interactive, with just one input.
    3. Label the input differently.
    4. Make the input more than just something to type in (a slider, list of options, etc.) if that is appropriate.
    5. Add a different control for something else - perhaps an option, a way to visualize it, etc.
    6. Add more controls until you love your interact!